Uniform Evaluation of Hardware Implementations of the Round-Two SHA-3 Candidates

Stefan Tillich[1,2], Martin Feldhofer[1], Mario Kirschbaum[1], Thomas Plos[1], Jörn-Marc Schmidt[1], and Alexander Szekely[1]

[1] Graz University of Technology, Institute for Applied Information Processing and Communications, Inffeldgasse 16a, A–8010 Graz, Austria
{Stefan.Tillich,Martin.Feldhofer,Mario.Kirschbaum, Thomas.Plos,Joern-Marc.Schmidt,Alexander.Szekely}@iaik.tugraz.at
[2] University of Bristol, Computer Science Department, Merchant Venturers Building, Woodland Road, BS8 1UB, Bristol, UK
tillich@cs.bris.ac.uk

Abstract. We describe our high-speed hardware modules for the 14 candidates of the second evaluation round of the SHA-3 hash function competition. Emphasis has been put on bringing as many aspects of design and implementation as possible into agreement in order to receive consistent and comparable evaluation results. For most candidates we have tested a range of different design and implementation options. The evaluation involved a large number of synthesis runs in a uniform setup and under the use of a simple optimization heuristic. In addition to identifying good hardware-design options, this approach has yielded data on numerous possible area-performance tradeoffs for the different hardware modules. The best configurations then underwent place & route in order to reach the highest degree of accuracy of performance metrics short of actual implementation in silicon.

Keywords: SHA-3, hardware, ASIC, standard-cell implementation, high speed, high throughput, BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, Skein.

1 Introduction

Following the weakening of the widely-used SHA-1 hash algorithm and concerns over the similarly-structured algorithms of the SHA-2 family, the US NIST has initiated the SHA-3 contest in order to select a suitable drop-in replacement [23]. In round two of the competition, 14 candidates remain for consideration. Apart from the ongoing cryptanalytic efforts, benchmarking of software and hardware implementations of the candidates will be an important part of the evaluation. Software benchmarking is done for example by the NIST on their reference platform and by the eBASH project in the context of the ECRYPT II network of excellence [4].

While a fair comparison of software performance is far from trivial, the situation for hardware implementations seems even worse. Although hardware-performance figures for several candidate algorithms have been published [12], a fair and meaningful comparison of these results is extremely difficult. Hardware modules are designed towards different goals (e.g. maximal throughput, low area, optimal speed-area tradeoff) and feature varying degrees of functionality and system interfaces. Moreover, implementation often involves different synthesis tools, target technologies, and optimization heuristics.

The Athena project has the goal of allowing a more meaningful comparison of hardware performance on FPGAs [16]. This effort is aided by the relatively broad availability of corresponding design and synthesis tools (e.g. Xilinx ISE, Altera Quartus II) and the ease of using certain families of FPGAs as uniform target devices. For standard-cell hardware implementations, the availability of corresponding synthesis setups tends to be much more limited. Even if

the same tools are employed, the standard-cell libraries and target technologies seldom agree. Another problem is that the HDL code of implementations is not always made available, which often precludes benchmarking of implementations from different parties.

By designing and implementing high-speed hardware modules of all 14 candidate hash algorithms from scratch we were able to overcome most of these problems. Our implementations encompass equivalent functionality and interfaces and received similar optimization effort.

We have concentrated on those hash function variants which produce a 256-bit message digest. Some of our hardware modules can additionally be reconfigured statically or dynamically to produce digests of a different size. Extra functionality like salting or keyed hashing modes are not supported. Furthermore, we have chosen to evaluate the performance of the hash functions themselves and not that of some arbitrary interface. To this end, we did not put any restrictions on the throughput for data which is fed into the hardware module. Nevertheless, we would like to stress that a candidate's ability to reach peak performance with a narrower data interface should be counted as an advantage.

Our hardware modules expect to receive padded messages (*i.e.* a number of full message blocks) as input. Resulting benefits are a simplified design and the possibility of a uniform interface which does not introduce communication overheads. As our primary concern is maximizing throughput for long messages, padding performed outside of the hardware hash module will have no detrimental effect on the peak throughput. Apart from padding, the hardware modules are fully self-contained and require no additional components (e.g. external memory).

Implementation has been done targeting the same process technology and the same standard-cell library and has made use of a uniform optimization heuristic. Our work is the first to provide a comparison of all 14 candidates and should serve as a starting point for a fair and transparent evaluation of the hardware performance of the SHA-3 candidates. In this context, our work is the first to present concrete hardware implementations of JH, SHAvite-3, and SIMD, and complete standard-cell implementations of BLAKE, Blue Midnight Wish, CubeHash, and Shabal.

The remainder of this paper is structured as follows. Section 2 gives a brief description of each candidate's hardware implementation. Section 3 explains our evaluation methodology and summarizes our results. A short conclusion is given in Section 4.

2 Description of the SHA-3 Hardware Modules

Due to space restrictions we concentrate on descriptions of our implementations. For details on the SHA-3 candidates please consult the respective specification documents and accompanying webpages.

2.1 BLAKE

We have implemented various design approaches of BLAKE-32 [1]. The best performance was obtained by implementing four parallel instances of the transformation operation G, which leads to two clock cycles for each round of BLAKE-32. Carry-save adders are used inside the G operations to speed up the computation. We have added a pipeline register at the output of the permutation table in order to reduce the critical path of the design. Moreover, delaying the finalization step, which produces the next chaining value, by one clock cycle additionally increases the performance. Figure 1 shows the datapath of this implementation.

2.2 Blue Midnight Wish

We have tested two basic implementation approaches of Blue Midnight Wish [17]: A pipelined implementation with shared adders/subtractors and a full unrolling of the message expansion and compression function. While the pipelined implementation is smaller, the fully unrolled module has a considerably higher throughput. The datapath of the hardware module with full unrolling is shown in Figure 2.

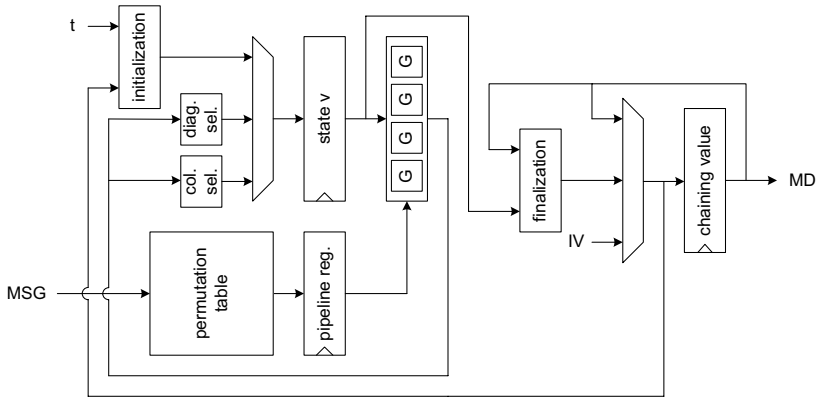

Fig. 1. Implementation of BLAKE-32.

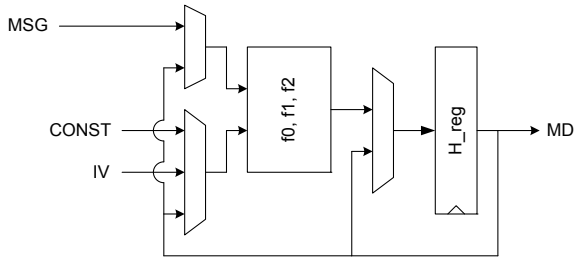

Fig. 2. Implementation of BMW-256 (fully unrolled).

2.3 CubeHash

Our implementation has been kept flexible, in order to accommodate a wide range of CubeHash [3] variants. The algorithm's parameters can be selected individually at runtime for each hashing operation. The full range of message-digest sizes h and message-block sizes b is supported. Furthermore, the number of rounds r can be configured up to a maximum value of 32 rounds. These parameters are read in the initialization phase of the hashing operation. The number of unrolled full CubeHash rounds is statically configurable prior to synthesis of the design. The control finite-state machine adapts flexibly to the runtime parameters and the number of unrolled rounds.

The core of our implementation is constituted by the 1,024-bit state and a combinatorial unit consisting of the configured number of CubeHash rounds. The clock-cycle latency per message block is simply the number of rounds r divided by the number of unrolled rounds. Initialization and finalization take $10r+1$ cycles, respectively. If an extra cycle is allowed for the loading of message blocks, the area of the implementation can be reduced at the cost of a lower throughput. The datapath of our CubeHash implementation with two unrolled rounds is depicted in Figure 3.

We have synthesized our CubeHash module with different numbers of unrolled rounds (1, 2, 4, or 8). Implementation with 16 unrolled rounds failed due to problems of the synthesis software with larger designs. In any case, increased degrees of unrolling turned out to be an ineffective measure to reach higher throughputs.

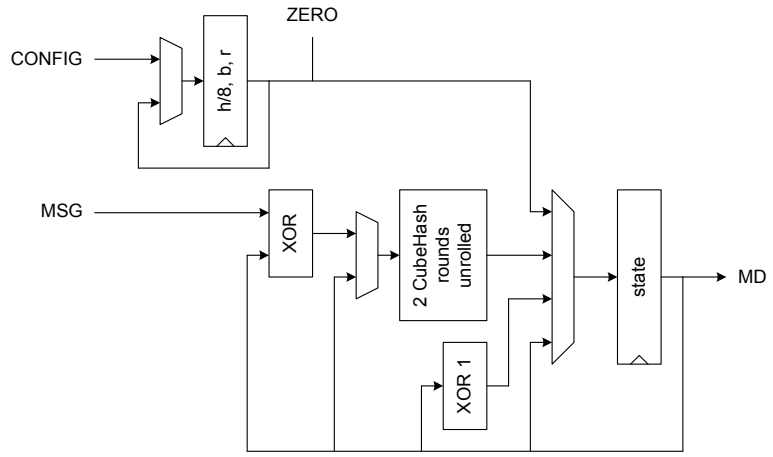

Fig. 3. Implementation of CubeHash.

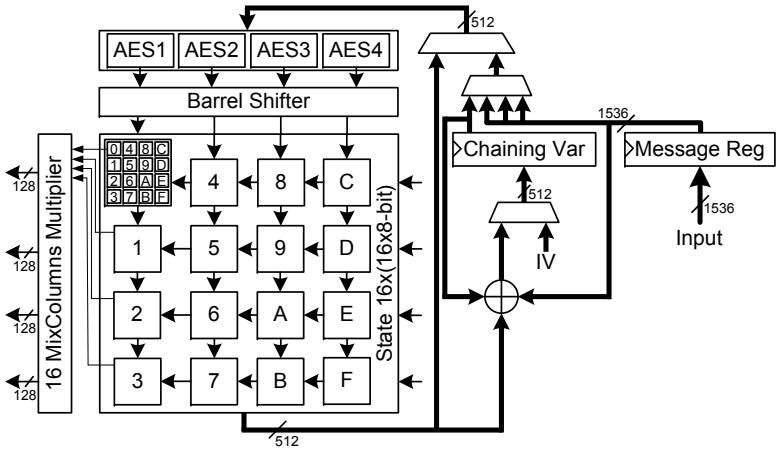

Fig. 4. Implementation of ECHO-256.

2.4 ECHO

The hardware implementation of ECHO-256 [2] is shown in Figure 4. The underlying architecture is similar to the AES implementation of Mangard *et al.* [21]. The central element is the State matrix which consists of 16×128-bit words (which are internally organized as 16×8-bit). We instantiated a whole AES round four times which makes up the largest combinational circuit of the hardware module. This allows to compute the two AES rounds for the BIG.SubWords operation for each 128-bit word in eight clock cycles. Additional four clock cycles are necessary for calculating BIG.MixColumns whereby 16 instances of AES MixColumns multipliers are used. For the total of eight rounds, this leads to a latency of 96 clock cycles. One additional clock cycle is required for the BIG.Final operation at the end of hashing a 1,536-bit input block.

For our module we investigated the use of several different implementations of the AES S-box, resulting in a trade-off between size and throughput. Alternatively, it would be possible to achieve further speed-up by using 64 MixColumns multipliers instead of 16 or to instantiate 16 parallel AES rounds instead of four. This would reduce the number of clock cycles by 24 and 48 respectively, but would in turn increase the required area considerably.

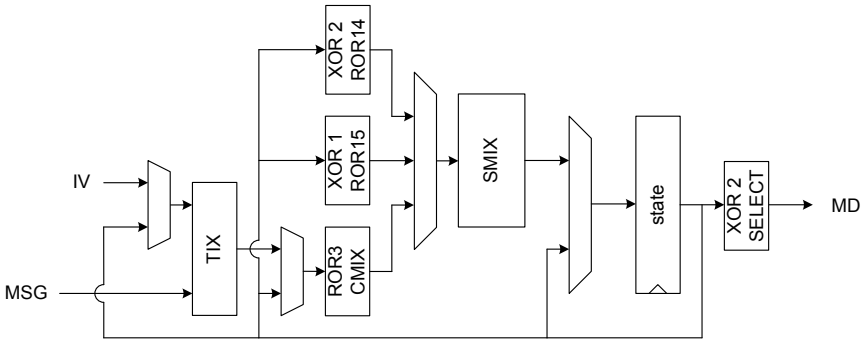

Fig. 5. Implementation of Fugue-256.

2.5 Fugue

Our implementation of Fugue-256 [18] is depicted in Figure 5. The TIX operation has been integrated in the loading operation of the message block in order to keep the number of clock cycles per block small. In our case, a 32-bit block is processed in two clock cycles.

The "heaviest" operation of Fugue is its SMIX transformation. As SMIX resembles parts of the AES round, similar optimization techniques apply. The whole transformation can be implemented as 16 parallel look-ups of 128-bit values, with a subsequent combination of the 16 values into the final output (similar to the T-table approach in AES [11]). Alternatively, the S-box layer can be implemented separately from the matrix multiplication. We have tested both approaches. For the case of separated S-box look-up and matrix multiplication, we have compared two different implementations of the AES S-box: Canright's solution using normal bases [10] and a synthesized hardware look-up table of the input-output mapping (HW LUT approach) [25].

2.6 Grøstl

For our high-speed Grøstl-256 [15] implementation we have tested both the parallel calculation of the P and Q permutation as well as the use of a single permutation unit which can switch between both permutation types. Interestingly, the second possibility has a higher throughput, even though it is much smaller than the first. This is due to the fact that the throughput of a design with two parallel permutations cannot be increased further by inserting pipeline stages within rounds due to the inherent data dependencies of the intermediate results.

Our fastest implementation features a pipelined permutation round unit with two stages. This unit is used to calculate the P and Q permutation alternatingly. Two 512-bit registers are used to hold intermediate results and the previous chaining value, respectively. This version is shown in Figure 6.

2.7 Hamsi

We have investigated two versions of hardware implementations of Hamsi-256 [19] which differ in the number of instances of the non-linear permutation function P and Pf. The state matrix is stored in a 512-bit register and the chaining value requires a 256-bit register.

A P/Pf instance mainly consists of 128 S-boxes, which are implemented as an unstructured mass of standard cells (HW LUT approach [25]), and four L transformation modules. Additionally, round constants and the round counter are added using XOR gates. The truncation function

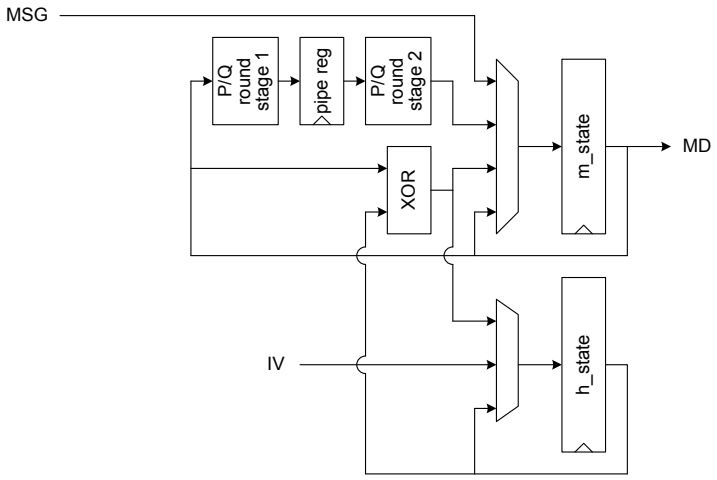

Fig. 6. Implementation of Grøstl-256.

is simply realized as rewiring and the feed forward of the chaining value is an XOR operation of the truncated state with the previous chaining value. The message expansion is implemented as a table lookup which is quite efficient.

The architecture of our fastest implementation of Hamsi-256 is depicted in Figure 7. It features three instances of the *P*/*Pf* function and requires one clock cycle to hash a 32-bit block (except for the last block, which requires two cycles).

2.8 JH

Our implementation of JH-256 [27] works with 320 instances of a combinational implementation of the S-boxes; 256 S-boxes transform the internal state H and 64 S-boxes work on the round constant vector C_r in every round of R_8. This way, one round of R_8 can be executed in only one clock cycle. The datapath of our implementation is illustrated in Figure 8.

In our implementation of JH the registers for the 1024-bit internal state H, the 512-bit message block M_i, and the 256-bit round constant C_r occupy approximately one quarter of the whole area. The 320 combinational S-boxes occupy another quarter of the area. One 512-bit message block is processed in 39 clock cycles.

2.9 Keccak

The plain structure of Keccak [5] naturally maps to the simple implementation depicted in Figure 9. Through static configuration, our implementation supports all variants of the Keccak hash function which have been proposed as SHA-3 candidates. For the performance evaluation we concentrated on the 256-bit variant, namely $\lfloor \text{Keccak}[r{=}1088, c{=}512, d{=}32] \rfloor_{256}$[3].

A single round of the Keccak-f permutation is instantiated in hardware. Thus, a total of 24 iterations is required to perform the complete permutation. The appropriate round constant is selected by the current round index. As the round constants have a very low Hamming weight, they can be mapped to a small synthesized look-up table.

[3] In this notation, r denotes the message block size in bits, c is the state size (fixed to 1,600 bits) minus the block size, and d is the so-called diversifier, which is used in message padding. The $\lfloor \rfloor_{256}$ notation indicates truncation of the state to 256 bits in order to generate the message digest

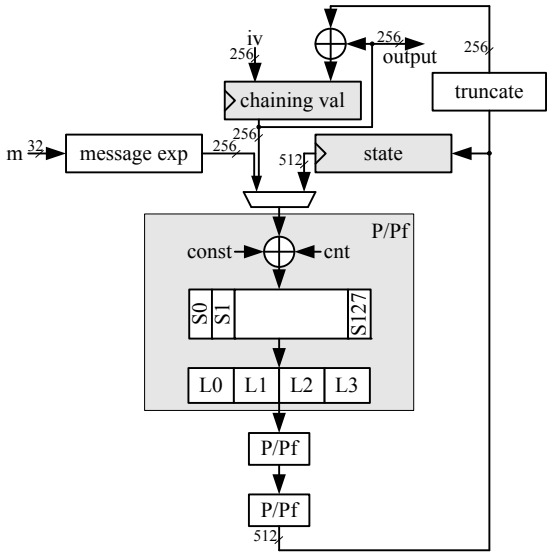

Fig. 7. Architecture overview of fastest Hamsi-256 implementation.

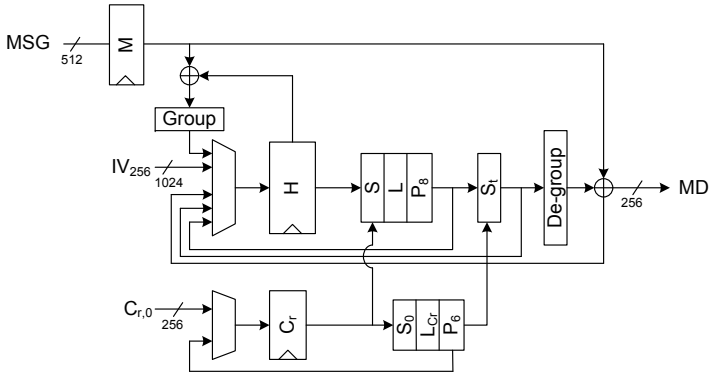

Fig. 8. Implementation of JH-256.

The loading of the message block and its combination with the state requires an additional clock cycle. This separation allows to reduce the critical path of the hardware module, which runs through the Keccak-f round unit. The processing of a complete message block thus requires 25 clock cycles.

2.10 Luffa

As the two smallest variants of Luffa [9] (Luffa-224 and Luffa-256) are virtually identical, we have implemented a hardware module capable of supporting both variants. The corresponding datapath is shown in Figure 10. The inputs for the message injection function can be switched to accommodate the first message block (IV and MSG loaded), intermediate message blocks (state feedback and MSG loaded), and final blank rounds (state feedback and ZERO loaded). The tweak at the start of each permutation is already performed at the end of the message injection. The constants are generated on-the-fly and the current constants are registered in

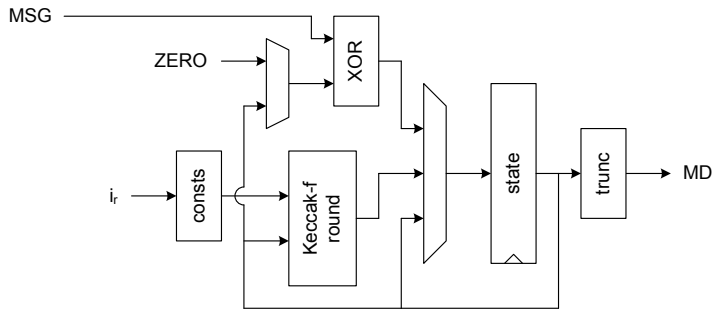

Fig. 9. Implementation of Keccak.

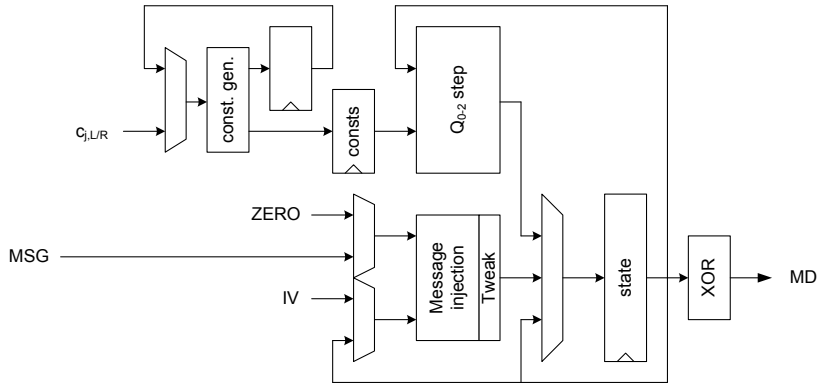

Fig. 10. Implementation of Luffa-224/256.

order to minimize the critical path. One step for each of the three permutations Q_0, Q_1, and Q_2 is implemented in parallel.

Luffa consists of rather simple operations which can be mapped efficiently to hardware (simple bit-sliced S-boxes, fixed rotations, XORs, and arithmetic operations in binary extension fields). By separating message injection from the Q_j steps, the combinatorial paths can be split up relatively evenly. The message injection has been implemented following the approach given in the specification [9], which uses doubling of GF(2^{32}) elements as basic building block. For the 4-bit S-box layer, we have implemented both the bit-sliced approach as well as explicit instantiation of the S-boxes as synthesized look-up tables (both resulting in similar speed).

2.11 Shabal

Figure 11 depicts the datapath of our implementation of Shabal [7]. It basically consists of 32-bit adders, a 384-bit shift register for A and three 512-bit shift registers for B, C and M. Each round of the permutation P rotates A, B, C, and M by one 32-bit word. Furthermore, the results of the combinatorial logic are put on the last position of A and B. Since there are 48 rounds of P, each register is in the correct position after the application of P (A is fully rotated four times, B and C are fully rotated three times). The initialization vectors are stored as constants, which saves two initial rounds. Each inner round requires one clock cycle for adding before P and one cycle for the subtraction after P. Each of the 48 inner loops of P requires one cycle, resulting in a total latency of 50 cycles per message block.

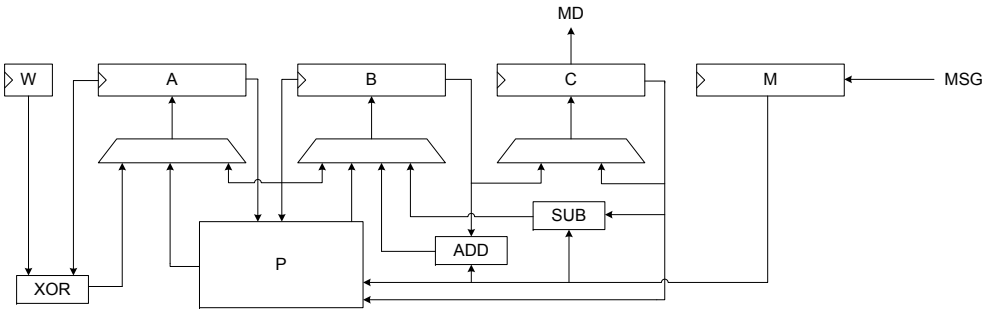

Fig. 11. Implementation of Shabal-256.

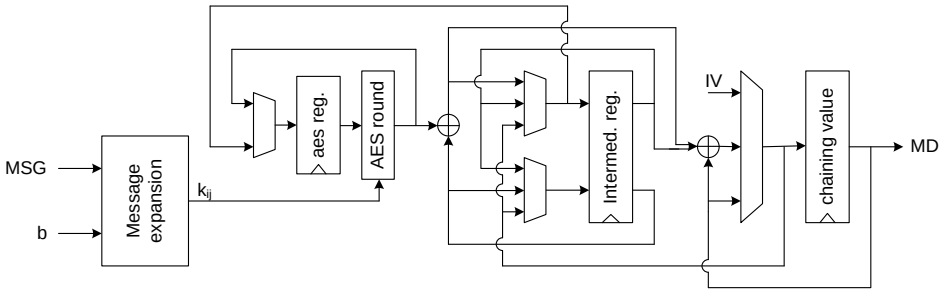

Fig. 12. Implementation of SHAvite-3_{256} with one AES round.

2.12 SHAvite-3

Several versions of SHAvite-3_{256} [6] have been implemented in hardware and evaluated with respect to their performance. The versions mainly differ in the number of unrolled AES rounds for the compression function and the message expansion. Additionally, various implementations of the AES-round function have been tested: The T-table approach [11], and separated S-box and MixColumns layers, with S-box implementations following Canright [10], Wolkerstorfer *et al.* [26], and the HW LUT approach [25].

The architecture which yielded the highest throughput in our evaluation is depicted in Figure 12. It features one AES round for the compression function and one AES round for the message expansion (contained in the *message expansion* block). The S-boxes were implemented following the HW LUT approach.

2.13 SIMD

Our implementation of SIMD-256 [20] realizes the specification of round 1 of the SHA-3 competition. The hardware module is depicted in Figure 13. It consists of four Feistel blocks in parallel. Implementing the number-theoretic transform (NTT) modulo 257 of the 64 input bytes basically means performing a Fast Fourier Transform (FFT) mod 257 of 128 integer values. As half of these 128 values are zero, this FFT can be split into two separate FFT-64. Each FFT-64 is built from two instances of FFT-8 and sixteen 8×8-bit modulo multipliers. With this configuration we need 36 clock cycles to process a 512-bit message block.

2.14 Skein

We have implemented Skein [14] with all three block sizes for Threefish (256, 512, and 1,024 bits). The core of the datapath consists of eight unrolled rounds of Threefish and a key schedule

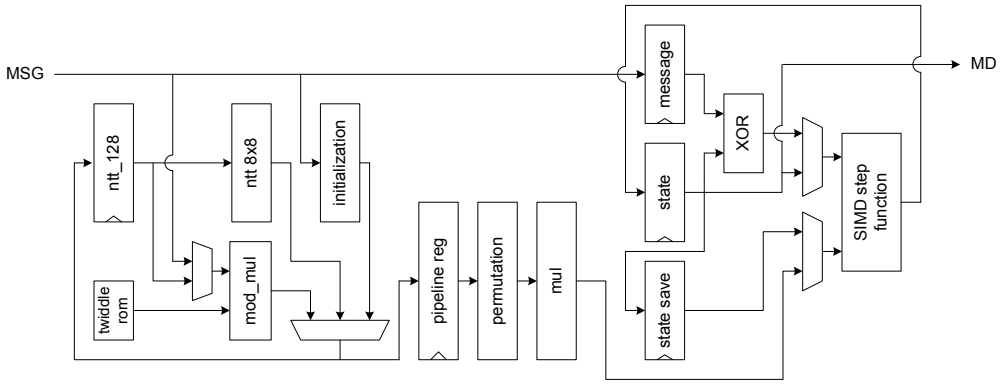

Fig. 13. Implementation of SIMD-256.

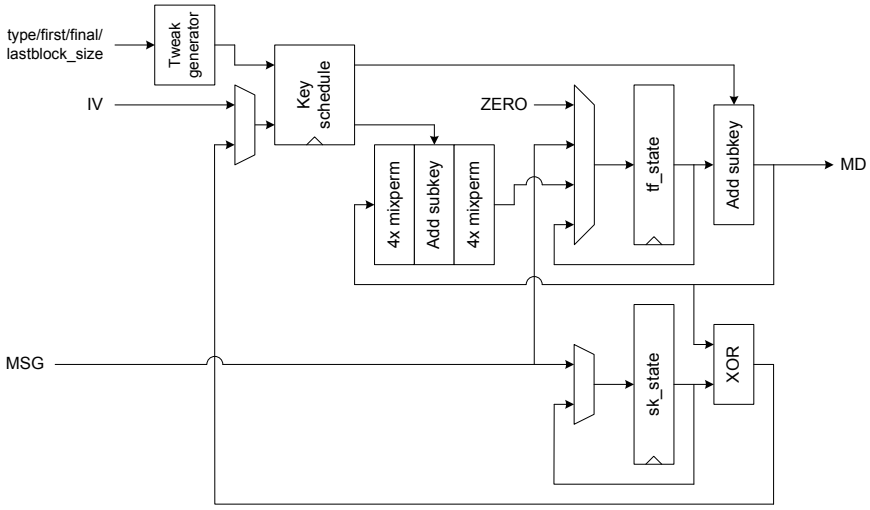

Fig. 14. Implementation of Skein.

unit which can supply two consecutive subkeys at a time. The advantage of this architecture is that the Threefish rounds have fixed rotation distances for their MIX layer, which allows a simple hard-wiring of the rotations. Thus, the output of the Threefish unit only depends on the input block and the two subkeys. Our implementation is shown in Figure 14.

The key schedule unit is loaded with an input key and input tweak at the beginning of each Threefish encryption. Two subsequent subkeys are derived through a number of 64-bit adders. Apart from the key schedule unit, the datapath contains two registers of the size of a Threefish block for the current message block and for holding intermediate values of the Threefish encryption.

2.15 SHA-2 Reference Implementation

To serve as a point of reference, we have implemented a SHA-256 [24] hardware module with a straight-forward approach. No optimization techniques [22] have been employed, except for the use of carry-save adders in the round implementation of compression function and message schedule. The datapath of our implementation is depicted in Figure 15.

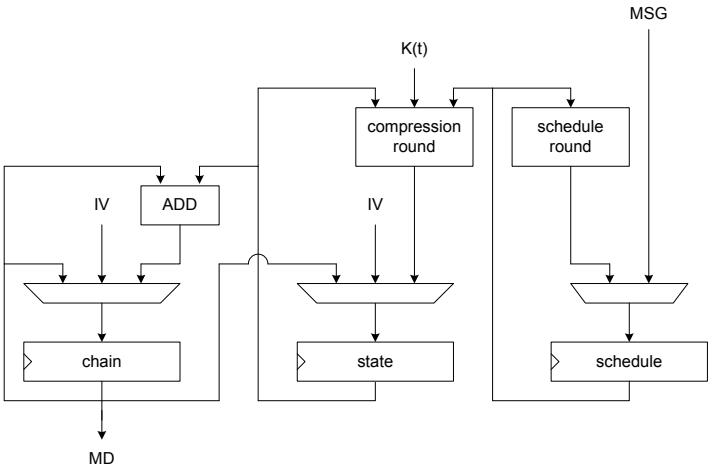

Fig. 15. Straight-forward implementation of SHA-256.

3 Practical Results

Shabal and SIMD have been implemented in Verilog; all other SHA-3 hardware modules have been written in VHDL. The HDL sources of our implementations will be made available to interested parties involved in the SHA-3 evaluation process upon request from one of the authors. Any eventual second-round tweaks have been integrated in the modules. The only exception is the SIMD module, which (due to time constraints of the authors) implements the specification from round one[4]. The results of the Skein-512-512 module are reported as an estimation for a Skein-512-256 hardware module, as the specification seems to allow both Skein-256-256 and Skein-512-256 as SHA-3 variant with a 256-bit message digest. Correct functionality of all modules has been verified against the official Known Answer Test (KAT) vectors with simulation via Cadence NCSim [8].

Our throughput evaluation assumes that the message blocks are delivered to the hardware module at a speed which allows it to operate under full utilization. Our optimization target for synthesis was *maximum peak throughput*, which corresponds to the throughput for long messages. Note that for shorter messages, the throughput might change due to more or less costly initialization operations and output transformations.

As first phase of evaluation we performed multiple standard-cell synthesis runs for each of the variants of the hardware modules using an adaptive optimization heuristic. For each run, the target for the critical path delay has been adapted[5]. Synthesis runs have been counted as successful only if they (1) finished within a certain amount of time[6] and (2) the synthesized design reached the set target delay under worst-case conditions[7].

In the second evaluation phase, the variants with the "best" performance were subjected to place & route. The primary selection criterion was high throughput. If throughput could only be increased in the range of a few percent at the cost of area increase in the range of several dozen percent, we chose the slightly slower but substantially more area-efficient implementation for

[4] We don't expect the integration of the round-two tweak to have a significant effect on the performance figures.

[5] Lowered if the run was successful, increased if it failed.

[6] For the present work, the limit has been set to two hours.

[7] A maximal negative slack of 50 ps has been allowed.

place & route. For example, a 4% increase in throughput at a 65% increase in area was not considered.

Synthesis and place & route targeted the UMC 0.18 µm standard-cell library FSA0A_C from Faraday [13] and has been performed with the Cadence PKS-Shell (v05.16) and Cadence First Encounter (v05.20), respectively [8]. Optimization effort was set high and was primarily aimed towards maximum speed.

Note that we only make a comparison of the results of our hardware modules and that we do not include previously published results. We do this in order to stress the coherency of our benchmarking effort and to keep the comparison as fair as possible.

The implementations reported in Table **??** refer to the following implementation variants:

- **BLAKE**: Four G functions in parallel, two pipeline registers, additional cycle for chaining, carry-save adders.
- **Blue Midnight Wish**: Whole compression function (f_0, f_1, f_2) as a single combinational block, generic adders.
- **CubeHash**: Two CubeHash rounds unrolled, generic adders.
- **ECHO**: S-boxes as HW LUT.
- **Fugue**: S-boxes and matrix multiplication separated, S-boxes as HW LUT.
- **Grøstl**: Shared P/Q permutation, S-boxes and MixBytes separated, S-boxes following Wolkerstorfers *et al.* approach [26] with one pipeline register.
- **Hamsi**: Three P/Pf instances in series, S-boxes as HW LUT.
- **JH**: 320 S-boxes (one cycle per R_8 round), combinational S-boxes.
- **Keccak**: One Kekkak-f round per cycle.
- **Luffa**: S-boxes and matrix multiplication separated, S-boxes as HW LUT, output of control FSM registered.
- **Shabal**: One round of permutation P per cycle, generic adders.
- **SHAvite-3**: One AES round for compression function and message expansion each.
- **SIMD**: Message expansion with 16 parallel FFT-8 and 16 parallel modular multipliers, compression function with four parallel Feistel blocks.
- **Skein**: Eight Threefish rounds unrolled, generic adders.
- **SHA-2**: No unrolling or quasi-pipelining, generic adders.

The estimations of the synthesis tool after synthesis differ from the results after place & route. Most importantly, the maximum clock frequency of most implementation is decreased, most likely due to routing overheads not anticipated by the synthesis tool. This leads in turn to lower throughput figures. Also, the gate count after place & route decreases slightly for most implementations. This is due to the removal of extra buffers from the post-synthesis netlist by the place & route tool.

Table 1 summarizes our results after place & route. It contains the block size of the hash algorithm (block) and the number of clock cycles required for the processing of one block (latency). The area is given in terms of gate equivalents (GEs)[8]. The reported clock frequency is the maximum value *under typical conditions*[9]. The throughput column indicates the peak throughput at the stated clock frequency. The block size and the latency determine the properties of the interface required to reach peak throughput. A large block size and a small latency (e.g. for the BMW-256 implementation) require either a very broad interface or an interface clocked substantially faster than the hash module. Appendix A contains a graphical representation of area in relation to highest throughput in Figure 16 and the results of the "best" implementation variants after synthesis alone in Table 2. With the exception of BMW, the maximum clock frequency (and it turn the throughput) of all implementations was lower after place & route than estimated after synthesis. On the other hand, the total area of most implementations decreased after place & route. Unfortunately, we did not have sufficient time to investigate the reasons for these differences.

[8] For FSA0A_C, 1 GE equals 9.37 sqmils (*i.e.* the size of a ND2 cell).

[9] Operating temperature 25 °C, supply voltage 1.8 V.

Table 1. Results after place & route for the "best" implementation variants using the UMC 0.18 µm FSA0A_C standard-cell library.

Implementation	Block bit	Latency cycles	Area GE	Clock freq. MHz	Throughput Gbit/s
BLAKE-32	512	22	38,877	144.15	3.355
BMW-256	512	1	160,922	15.12	7.741
CubeHash16/32-*h*	256	8	56,612	111.06	3.554
ECHO-256	1,536	97	128,069	121.97	1.931
Fugue-256	32	2	48,401	161.19	2.579
Grøstl-256	512	22	53,680	202.47	4.712
Hamsi-256	32	1	59,955	119.77	3.833
JH-256	512	39	51,212	259.54	3.407
Keccak(-256)	1,088	25	56,713	267.09	11.624
Luffa-224/256	256	9	45,271	336.02	9.558
Shabal-256	512	50	55,143	216.83	2.220
SHAvite-3_{256}	512	37	59,822	159.80	2.211
SIMD-256	512	36	95,669	58.33	0.830
Skein-256-256	256	10	47,678	64.75	1.658
Skein-512-512	512	10	76,250	43.49	2.227
SHA-256	512	66	19,515	211.37	1.640

In terms of throughput, the Keccak implementation outperforms all other modules by a considerable margin. The Luffa module is second fastest and more compact. The next-best implementations are those of Grøstl, Hamsi, JH, and CubeHash which all have similar area requirements. The BMW module achieves similar throughput, but at considerably higher hardware cost. The implementations of Fugue and BLAKE are a bit slower, but also smaller. The Shabal and SHAvite-3 modules are slower and bigger. They achieve similar performance. The Skein-512 implementation follows next with a considerable hardware cost. The ECHO module achieves similar throughput, but requires more area. The Skein-256 module follows with a moderate size. Our implementation of SIMD is the slowest in the field. The straight-forward SHA-256 implementation has the smallest area and achieves a throughput which is rather at the low end of the spectrum.

4 Conclusions

In this work we presented our high-speed hardware implementations of all 14 round-two candidates of the SHA-3 contest. All hash modules have been designed and implemented towards the same optimization goal and evaluated with the same synthesis tools, target technology, and optimization heuristic. In order to stress the coherency of our results, we have consciously excluded other prior published implementations from consideration, as we regard the differences in interface design, standard-cell library, target technology, synthesis tools, and optimization effort to make meaningful comparisons extremely difficult. Our results indicate that Keccak and Luffa implementations offer the best throughput and also the best area/throughput tradeoff. Most of the other candidate implementations have a higher throughput than our SHA-256 reference implementation.

Acknowledgements. The work described in this paper has been supported by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. The information in this document is provided as is, and no guarantee or warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information as its sole risk and liability.

References

1. J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE, version 1.3. Available online at `http://131002.net/blake/blake.pdf`, 2008.
2. R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 Proposal: ECHO. Available online at `http://crypto.rd.francetelecom.com/echo/doc/echo_description_1-5.pdf`, February 2009.
3. D. J. Bernstein. CubeHash specification (2.B.1). Available online at `http://cubehash.cr.yp.to/submission/spec.pdf`, October 2008.
4. D. J. Bernstein and T. Lange. eBASH: ECRYPT Benchmarking of All Submitted Hashes. `http://bench.cr.yp.to/ebash.html`.
5. G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. KECCAK specifications, Version 2 – September 10, 2009. Available online at `http://keccak.noekeon.org/Keccak-specifications-2.pdf`, September 2009.
6. E. Biham and O. Dunkelman. The SHAvite-3 Hash Function (version from February 1, 2009). Available online at `http://www.cs.technion.ac.il/~orrd/SHAvite-3/Spec.01.02.09.pdf`, February 2009.
7. E. Bresson, A. Canteaut, B. Chevallier-Mames, C. Clavier, T. Fuhr, A. Gouget, T. Icart, J.-F. Misarsky, M. Naya-Plasencia, P. Paillier, T. Pornin, J.-R. Reinhard, C. Thuillet, and M. Videau. Shabal, a Submission to NIST's Cryptographic Hash Algorithm Competition. Available online at `http://www.shabal.com/wp-content/plugins/download-monitor/download.php?id=Shabal.pdf`, October 2008.
8. Cadence Design Systems. The Cadence Design Systems Website. `http://www.cadence.com/`.
9. C. D. Canniére, H. Sato, and D. Watanabe. Hash Function Luffa, Specification Ver. 2.0. Available online at `http://www.sdl.hitachi.co.jp/crypto/luffa/Luffa_v2_Specification_20090915.pdf`, September 2009.
10. D. Canright. A Very Compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.
11. J. Daemen and V. Rijmen. *The Design of Rijndael*. Information Security and Cryptography. Springer, 2002. ISBN 3-540-42580-2.
12. ECRYPT II. SHA-3 Hardware Implementations. `http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations`.
13. Faraday Technology Corporation. Faraday FSA0A_C 0.18 μm ASIC Standard Cell Library, 2004. Details available online at `http://www.faraday-tech.com`.
14. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein Hash Function Family. Available online at `http://www.skein-hash.info/sites/default/files/skein1.1.pdf`, November 2008.
15. P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, and S. S. T. Martin Schläffer. Grøstl – a SHA-3 candidate. Available online at `http://www.groestl.info/Groestl.pdf`, October 2008.
16. George Mason University. The Athena Website. `http://cryptography.gmu.edu/athena/`.
17. D. Gligoroski and V. Klima. Cryptographic Hash Function BLUE MIDNIGHT WISH. Available online at `http://people.item.ntnu.no/~danilog/Hash/BMW-SecondRound/Supporting_Documentation/BlueMidnightWishDocumentation.pdf`, September 2009.
18. S. Halevi, W. E. Hall, and C. S. Jutla. The Hash Function "Fugue". Available online at `http://domino.research.ibm.com/comm/research_projects.nsf/pages/fugue.index.html/$FILE/fugue_09.pdf`, September 2009.
19. Ö. Küçük. The Hash Function Hamsi, version from September 14, 2009. Available online at `http://www.cosic.esat.kuleuven.be/publications/article-1203.pdf`, September 2009.
20. G. Leurent, C. Bouillaguet, and P.-A. Fouque. SIMD Is a Message Digest. Updated version: 2009-01-15, 2009.
21. S. Mangard, M. Aigner, and S. Dominikus. A Highly Regular and Scalable AES Hardware Architecture. *IEEE Transactions on Computers*, 52(4):483–491, April 2003.
22. R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane. Optimisation of the SHA-2 Family of Hash Functions on FPGAs. In J. Becker, A. Herkersdorf, A. Mukherjee, and A. Smailagic, editors,

IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06), Karlsruhe, Germany, 2-3 March, 2006, Proceedings, pages 317–322. IEEE Computer Society, March 2006.

23. National Institute of Standards and Technology (NIST). Cryptographic Hash Algorithm Competition Website. `http://csrc.nist.gov/groups/ST/hash/sha-3`.
24. National Institute of Standards and Technology (NIST). FIPS-180-3: Secure Hash Standard, October 2008. Available online at `http://www.itl.nist.gov/fipspubs/`.
25. S. Tillich, M. Feldhofer, and J. Großschädl. Area, Delay, and Power Characteristics of Standard-Cell Implementations of the AES S-Box. In S. Vassiliadis, S. Wong, and T. Hämäläinen, editors, *6th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2006, Samos, Greece, July 17-20, 2006, Proceedings*, volume 4017 of *Lecture Notes in Computer Science*, pages 457–466. Springer, July 2006.
26. J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES SBoxes. In B. Preneel, editor, *Topics in Cryptology - CT-RSA 2002, The Cryptographers' Track at the RSA Conference 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings*, volume 2271 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2002.
27. H. Wu. SHA-3 proposal JH, version January 15, 2009. JH online at http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/index.html, 2008.

A Further Practical Results

Table 2. Results after synthesis for the "best" implementation variants using the UMC 0.18 µm FSA0A_C standard-cell library.

Implementation	Block bit	Latency cycles	Area GE	Clock freq. MHz	Throughput Gbit/s
BLAKE-32	512	22	45,640	170.64	3.971
BMW-256	512	1	169,737	10.46	5.358
CubeHash16/32-*h*	256	8	58,872	145.77	4.665
ECHO-256	1,536	97	141,489	141.84	2.246
Fugue-256	32	2	46,257	255.75	4.092
Grøstl-256	512	22	58,402	270.27	6.290
Hamsi-256	32	1	58,661	173.91	5.565
JH-256	512	39	58,832	380.22	4.992
Keccak(-256)	1,088	25	56,316	487.80	21.229
Luffa-224/256	256	9	44,972	483.09	13.741
Shabal-256	512	50	54,186	320.51	3.282
SHAvite-3_{256}	512	37	57,388	227.79	3.152
SIMD-256	512	36	104,166	64.93	0.924
Skein-256-256	256	10	58,611	73.52	1.882
Skein-512-512	512	10	102,039	48.87	2.502
SHA-256	512	66	19,144	302.11	2.344

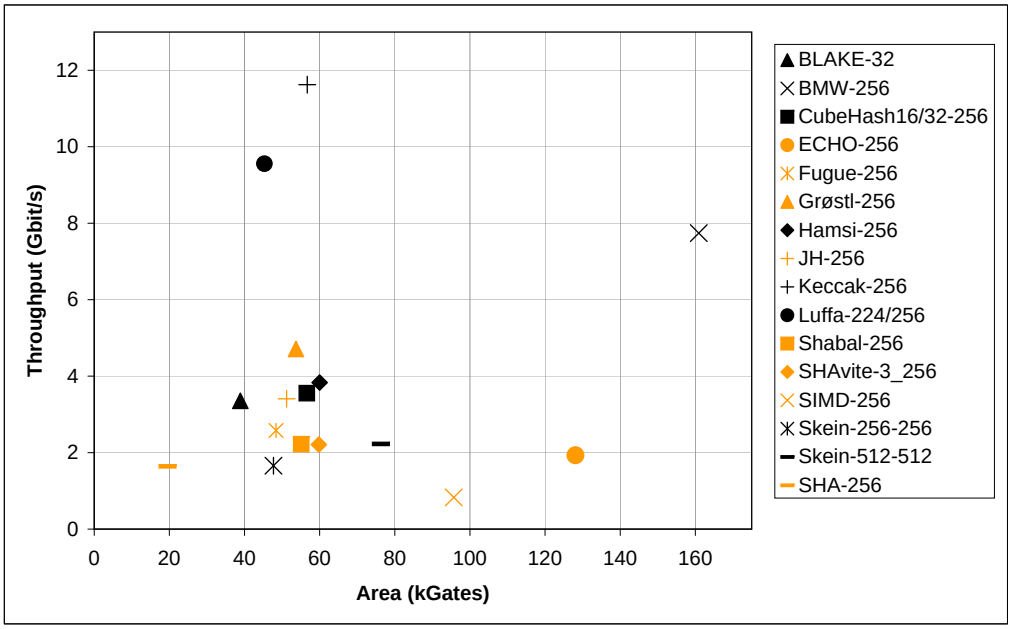

Fig. 16. Maximum peak throughput vs. area of the high-speed hardware implementations of the SHA-3 candidates (after place & route).